Building a Single Distributed File System from Many NFS Servers

Dan Muntz
Internet Systems and Storage Laboratory
HP Laboratories Palo Alto
HPL-2001-176
July 12th , 2001*

E-mail: dmuntz@hpl.hp.com

file system, distributed, NFS, virtualization

In this paper we describe an architecture, NFS^2, for uniting several NFS servers under a single namespace. This architecture has some interesting properties. First, the physical file systems that make up an NFS^2 instance, i.e., the file systems on the individual NFS servers, may be heterogeneous. This, combined with the way the NFS^2 namespace is constructed, allows files of different types (text, video, etc.) to be served from file servers (potentially) optimized for each type. Second, NFS^2 storage is strictly partitioned-each NFS server is solely responsible for allocating the resources under its control. This eliminates resource contention and distributed lock management, commonly found in cluster file systems. Third, because the system may be constructed with standard NFS servers, it can benefit from existing HA solutions for individual nodes, and improves as NFS servers improve.

* Internal Accession Date Only Approved for External Publication

Building a Single Distributed File System from Many NFS Servers

Dan Muntz

Hewlett-Packard Labs
1501 Page Mill Rd, Palo Alto, CA 94304, USA
dmuntz@hpl.hp.com

Abstract

In this paper we describe an architecture, *NFS^2*, for uniting several NFS servers under a single namespace. This architecture has some interesting properties. First, the physical file systems that make up an NFS^2 instance, i.e., the file systems on the individual NFS servers, may be heterogeneous. This, combined with the way the NFS^2 namespace is constructed, allows files of different types (text, video, etc.) to be served from file servers (potentially) optimized for each type. Second, NFS^2 storage is strictly partitioned—each NFS server is solely responsible for allocating the resources under its control. This eliminates resource contention and distributed lock management, commonly found in cluster file systems. Third, because the system may be constructed with standard NFS servers, it can benefit from existing HA solutions for individual nodes, and improves as NFS servers improve.

1 Introduction

NFS servers are widely used to provide file service on the Internet. However, adding new servers to an existing namespace is management intensive, and in some ways inflexible. When a new server is brought online, all clients requiring access to the new server must be updated to mount any new file systems from the server, and access rights for the new file systems must be configured on the server. Additionally, the new file systems are bound to sub-trees of each client's namespace.

The NFS^2 architecture allows standard NFS servers to be combined into a single, scalable file system. Each NFS server is essentially treated as an object store. New servers added to an NFS^2 system merely add more object storage—they are not bound to a particular location in the namespace. Clients accessing the NFS^2 file system need not be aware as new NFS servers are added or removed from the system.

2 Architecture

Figure 1 shows one possible configuration for an NFS^2 file system.

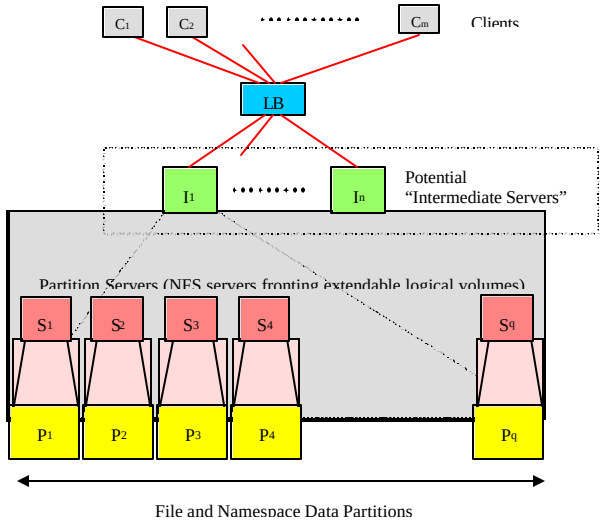

Figure 1: An NFS^2 File System

Storage partitions, P_i, are exported to the other parts of the system via standard NFS servers, S_i, also called *partition servers*. For scalability of the individual partitions/servers, *intermediate servers* can be introduced between the clients and servers. The intermediate servers accept NFS requests from the clients, and transform these requests into one or more NFS requests to the partition servers.

We describe the architecture under the assumption that the intermediate server translation functionality is embedded in the partition servers and that clients issue requests directly to the partition servers. An implementation based on this assumption would retain most of the benefits of the complete system (possibly sacrificing some ability to scale with single-file "hot spots"), but would also have some beneficial simplifications (e.g., reduced leasing overhead, fewer network hops, etc.).

3 Implementation

The most important concept behind the construction of the NFS^2 namespace is the *cross-partition reference*. A directory residing in one partition may have children (files or directories) residing on another partition.

There are a couple of alternatives for implementing cross-partition references in NFS^2. If we are allowed to modify the NFS servers, directories can be implemented as files in the underlying physical file system. This has advantages that will be explored in future work. With *directory files*, we can extend directory entries to support cross-partition references, independent of the physical file systems. To achieve the goal of using unmodified NFS servers, symbolic links can be used to construct cross-partition references. We first describe the system in terms of directory files (for clarity), and follow this with a description of how the same functionality can be achieved with symbolic links.

Another alternative is to store directories separately from files. Standard NFS servers are used to store files while separate servers are used for the namespace (either using directory files, or something else). Cross-partition references enable this separation of the namespace from the files. Servers for the namespace could be NFS servers modified to support directory files, a database, or some other construct.

The NFS^2 file system consists of user files and directory files. Both types of files exist as standard files in their respective partitions—a user file, `/usr/dict/words` might be represented as the file `/abc/123` on partition P_3, while the directory `/usr/dict` might be a file `/def/xxx` (containing *directory entries*) on partition P_4. An NFS^2 *directory entry* associates the user's notion of a file or directory name with the system's name for the file/directory, the partition where the file/directory is located, and any other relevant information. For example, some entries in `/def/xxx` could be represented as:

`.:/def/xxx:`P_4
`..:/yyy:`P_6
`words:/abc/123:`P_3

File handles passed to the client contain some representation of the system's name for the object (file or directory) and the partition where the object resides. This information is opaque to the client, but may be interpreted by the load balancer (LB) to direct requests to the correct partition server. Alternatively, the information could only be interpreted by partition servers, which may then have to forward the request one "hop" to the server responsible for the object.

In the initial state, a well-known root partition server (say, P_1) contains a file, e.g., "`/root`", which corresponds to the user's view of the root of the NFS^2 distributed file system. The client mounts the file system by obtaining a file handle for the `/root` file as a special case of the lookup RPC.

Let us consider how some operations are handled in this file system. A `mkdir` request from a client will contain a file handle for the parent directory (pfh) and a name for the new directory (dname). A *switch function* is used by LB to direct the request to the partition server (P_x) where the new directory will reside. The switch function arbitrarily defines a policy for where new file system objects are created. P_x creates a new file representing dname that has the name dname' in the physical file system served by P_x. P_x then issues a request to the partition server responsible for the parent directory, P_y (extracted from pfh), to add a directory entry: dname:dname':P_x to the parent directory file (contained in pfh). If an entry for dname already exists, the operation is aborted and dname is removed from P_x. Otherwise,

the new directory entry is added to the parent directory file and the operation completes.

The communication between P_x and P_y could be implemented using the standard NFS and lock manager protocols. P_x first locks the parent directory file, and checks for the existence of dname. If no entry for dname exists, it can issue an NFS write request to add the entry to the parent directory file. The directory file is subsequently unlocked. Alternatively, this communication could take place via a simple supplementary protocol that would allow the locking to be more efficient—a single RPC is sent to P_y, which then uses local file locking for the existence checking and update, and returns the completion status.

File creation is essentially identical to `mkdir`.

The `read` and `write` operations are trivial:

`write(fh, data, offset, length):`

- C_i sends the request to LB.
- LB looks into fh and directs the request to the appropriate S_j.
- S_j issues a local write call to the file specified in fh.

`Read` is similar.

To construct cross-partition references with symbolic links, we can build an NFS^2 cluster as a proof-of-concept as follows. First, each partition is assigned a name (assume Pi, as in Figure 1). The NFS servers then mount all partitions into their local namespace at locations /P1, /P2, etc. using the standard mount protocol. Now, a cross-partition reference is created by making a symbolic link that references the physical file through one of these mount points.

For the example:

`words:/abc/123:`P_3

An underlying file, `/abc/123`, contains the data for the file, and resides on partition P_3. The namespace entry `words` is a symbolic link in its parent directory with the link contents: `/P3/abc/123`.

4 Future Work

There are several areas requiring further investigation. Performance of the architecture in its various possible incarnations (the symbolic link version, the directory file version, and others) must be studied.

We also want to investigate the potential uses and performance implications of directory files. Directory files were conceived for the NFS^2 architecture to address the problem of providing a single directory structure over diverse underlying file systems, and the need for an easily extensible directory structure. Such benefits may be useful for other file system research. Also, because directory files allow flexibility of the directory structure, they could be used to investigate alternative data structures for directories, and alternative naming schemes.

Due to the structure of cross-partition references, object-level migration should be relatively straightforward in NFS^2. Migration and replication are two more areas requiring further research.

5 Related work

There has been a significant amount of research and product development in the area of cluster file systems [1,2,4]. Most are based on principles established in the VAXclusters [2] design. These systems use distributed lock management to control access to shared resources, which restrict their scalability. NFS^2 partitions resources to eliminate DLM.

Frangipani [1] uses a partitioning approach to shared resources at the volume level—each volume has its own DLM [3]. However, the file system implementation on top of the partitioned volumes is still limited by shared resource contention at the file system level.

6 Conclusions

NFS^2 provides a mechanism for uniting NFS servers under a single namespace. It simplifies management of multiple NFS servers by providing access to all servers through a single namespace (no need for

multiple client mount points), and by providing a transparent mechanism for the addition of new servers as the system grows.

This system avoids distributed lock management, which has been a limiting factor in the scalability of cluster file systems. NFS^2 supports heterogeneous physical file systems within the single namespace, whereas other systems have relied on their own proprietary physical file systems.

References

1. Thekkath, C., T. Mann, and E. Lee. *Frangipani: A Scalable Distributed File System.* In *16th ACM Symposium on Operating Systems Principles (SOSP)*, Saint-Malo, France, 1997.
2. Kronenberg, N., H. Levy, and W. Stecker, *VAXClusters: A closely-coupled distributed system.* ACM Tansactions on Computer Systems, 1986, 4(2): pp. 130-146.
3. Lee, E. and C. Thekkath. *Petal: Distributed Virtual Disks.* In *ASPLOS VII*, MA, USA, 1996.
4. *Veritas Cluster File System (CFS)*, 2000, Veritas Corp., Mountain View, California. http://www.veritas.com.